I. INTRO TO C

Hello World

```
#include <stdio.h>

int main(void) {
  printf("Hello World!\n");
  return 0;
}
```

- `#include`: preprocessor inserts `stdio.h` contents
- `stdio.h`: contains `printf` declaration
- `main`: program starts here
- `void`: keyword for argument absence
- `{ }`: basic block/scope delimiters
- `printf`: prints to the terminal
- `\n`: newline character
- `return`: leave function, return value

Compiling

```
$ gcc hello.c -o hello
$ ./hello
Hello World!
```

Basic Data Types

- `char c = 5; char c = 'a';`
  one byte, usually for characters (1970: ASCII is fine)
- `int i = 5; int i = 0xf; int i = 'a';`
  usually 4 bytes, holds integers
- `float f = 5; float f = 5.5;`
  4 bytes floating point number
- `double d = 5.19562`
  8 bytes double precision floating point number

Basic Data Types – logic

- `int i = 5 / 2; //i = 2`
  integer logic, no rounding
- `float f = 5.0f / 2; //f = 2.5f`
  decimal logic for float and double
- `char a = 'a'/ 2 //a = 97 / 2 = 48`
  `char` interpreted as character by console

Basic Data Types – signed/unsigned

- `signed int i = -5 //i = -5` (two's complement)
- `unsigned int i = -5 //i = 4294967291`

Basic Data Types – short/long

- `short int i = 1024 //-32768...32767`
- `long int i = 1024 //-2147483648...2147483647`

Basic Data Types – more size stuff

- `sizeof int; sizeof long int; //4; 4;` (x86 32-Bit)
- use data types from `inttypes.h` to be sure about sizes:

```
#include <inttypes.h>
int8_t i; uint32_t j;
```

Basic Data Types – const/volatile

- `const int c = 5;`
  `i` is constant, changing it will raise compiler error
- `volatile int i = 5;`
  `i` is volatile, may be modified elsewhere (by different program in shared memory, important for CPU caches, register, assumptions thereof)

Variables – local vs. global

```
int m; // global variable

int myroutine(int j) {
  int i = 5 // local variable
  i = i+j;
  return i;
}
```

- *global variables* (`int m`):
  lifetime: while program runs
  placed on pre-defined place in memory
- basic block/function-local variables (`int i`):
  lifetime: during invocation of routine
  placed on stack or in registers

Variables – local vs. static

```
int myroutine(int j) {
  static int i = 5;
  i = i+j;
  return i;
}

k = myroutine(1); // k = 6
k = myroutine(1); // k = 7
```

- static function-local variables:
  saved like global variables
  variable persistent across invocations
  lifetime: like global variables

Printing

```
int i = 5; float f = 2.5;
printf("The numbers are i=%d, f=%f", i, f);
```

- comprised of format string and arguments
- may contain format identifiers (`%d`)
- see also `man printf`
- special characters: encoded via leading backslash:
  `\n` newline
  `\t` tab
  `\'` single quote
  `\"` double quote
  `\0` `null`, end of string

Compound data types

- `struct`ure: collection of named variables (different types)
- `union`: *single* variable that can have multiple types
- members accessed via `.` operator

```
struct coordinate {
  int x;
  int y;
}

union longorfloat {
  long l;
  float f;
}

struct coordinate c;
c.x = 5;
c.y = 6;

union longorfloat lf;
lf.l = 5;
lf.f = 6.192;
```

Functions

encapsulate functionality (*reuse*)
code structuring (*reduce complexity*)
must be **declared** and **defined**
Declaration: states signature
Definition: states implementation (implicitly declares function)

```
int sum(int a, int b); // declaration

int sum(int a, int b) { // definition
  return a+b;
}
```

Header files

header file for frequently used declarations
use **extern** to declare global variables defined elsewhere
use **static** to limit scope to current file (e.g. **static float** pi in sum.c: no pi in main.c)

```
// mymath.h
int sum(int a, int b);
extern float pi;

// sum.c
#include "mymath.h"

float pi = 3.1415927;
int sum(int a, int b) {
  return a+b;
}

// main.c
#include <stdio.h>
#include "mymath.h"

void main() {
  printf("%d\n", sum(1,2));
  printf("%f\n", pi);
}
```

Data Segments and Variables

Stack: local variables
Heap: variables crated at runtime via malloc()/free()
Data Segment: static/global variables
Code: functions

Function overloading

no function overloading in C!
use arrays ore pointers

Pointers

```
int a = 5;
int *p = &a // points to int, initalized to point to a
int *q = 32 // points to int at address 32
int b = a+1;
int c = *p; // dereference(p) = dereference(&a) = 5
int d = (*p)+2 // = 7
int *r = p+1; // pointing to next element p is pointing to
int e = *(p+2) // dereference (p+2) = d = 7
```

Pointers – linked list

linked-list implementation via next-pointer

```
struct ll {
  int item;
  struct ll *next;
}

struct l first;
first.item = 123;

struct ll second;
second.item = 456;
first.next = &second;
```

Arrays

= fixed number of variables *continuously laid out in memory*

```
int A[5]; // declare array (reserve memory space)
A[4] = 25; A[0] = 24; // assign 25 to last, 24 to first elem
char c[] = {'a',5,6,7,'B'} // init array, length implicit
c[64] = 'Z' // NO bounds checking at compile/run (may raise
      protection fault)

// declare pointer to array; address elements via pointer:
char *p = c;
*(p+1) = 'Z'; p[3] = 'B'; char b = *p; // = 'a'
```

Strings

= array of **char**s terminated by NULL:

```
char A[] = { 'T', 'e', 's', 't', '\0' };
char A[] = "Test";
```

declaration via pointer:
`const char *p = "Test";`

common string functions (string.h):
length: `size_t strnlen(const char *s, size_t maxlen)`
compare:
`int strncmp(const char *s1, const char *s2, size_t n);`
copy: `int strncpy(char *dest, const char *src size_t n);`
tokenize: `char *strtok(char *str, const char *delim);`
(e.g. split line into words)

Arithmetic/bitwise operators

arithmetic operators:
a+b, a++, ++a, a+=b, a–b, a––, ––a, a–=b, a*b, a*=b, a/b, a/=b, a%b, a%=b
logical operators:
a&b, a|b, a>>b, a<<b, a^b, ~a
difference pre-/post-increment:

```
int a = 5;
if(a++ == 5) printf("Yes"); // Yes
a = 5;
if(++a == 5) printf("Yes"); // nothing
```

operators in order of precedence:

```
(), [], ->, .
!, ++, --, +y, -y, *z, &=, (type), sizeof
*, /, %
+, -
<<, >>
<, <=, >, >=
==, !=
&
^
|
&&
||
?, :
=, +=, -=, *=, /=, %=, &=, ~=,=, «=, »=|
,
```

Structures

brackets only needed for multiple statements
if/else, **for**, **while**, **do-while**, **switch**
may use **break**/**continue**
switch: need break statement, otherwise will fall through

```
if(a==b) printf("Equal") else printf("Different");
for(i=10; i>=10; i--) printf("%d", i+1);
int i=10; while(i--) printf("foo");
int i=0; do printf("bar"); while(i++ != 0);

char a = read();
switch(a) {
  case '1':
    handle_1();
    break;
  default:
    handle_other();
    break;
}
```

Type casting

- explicit casting: precision loss possible
  `int i = 5; float f = (float)i;`
- implicit casting: if no precision is lost
  `char c = 5; int i = c;`
- pointer casting: changes address calculation
  `int i = 5; char *p = (char *)&i; *(p+1)= 5;`
- type hierarchy: „wider“/„shorter“ types
  `unsigned int` wider than `signed int`
  operators cast parameters to widest type
  Attention: assignment cast after operator cast

C Preprocessor

- modifies *source code* before compilation
- based on preprocessor *directives* (usually starting with `#`)
- `#include <stdio.h>`, `#include "mystdio.h"`:
  copies contents of file to current file
- only works with strings in source file
- completely ignores C semantics

Preprocessor – search paths

- `#include <file>`: system include, searches in:
  `/usr/local/include`
  `libdir/gcc/[target]/[version]/include`
  `/usr/[target]/include`
  `/usr/include`
  (target: arch-specific (e.g. i686-linux-gnu),
  version: gcc version (e.g. 4.2.4))
- `#include "file"`: local include, searches in:
  directory containing current file
  then paths specified by `-i <dir>`
  then in system include paths

Preprocessor – definitions

- defines introduce replacement strings (can have arguments, based on string replacement)
- can help code structuring, often leading to source code cluttering

```
#define PI 3.14159265
#define TRUE (1)
#define max(a,b) ((a > b) ? (a) (b))
#define panic(str) do { printf(str); for (;;) } while(0);

#ifdef __unix__
# include <unistd.h>
#elif defined _WIN32
# include <windows.h>
#endif
```

Preprocessor – predefined macros

- system-specific:
  `__unix__`, `_WIN32`, `__STDC_VERSION__`
- useful:
  `__LINE__`, `__FILE__`, `__DATE__`

Libraries

- = collection of functions contained in object files, glued together in dynamic/static library
- ex.: Math header contains declarations, but not all definitions
  $\rightsquigarrow$ need to link math library: `gcc math.c -o math -lm`

```
#include <math.h>
#include <stdio.h>

int main() {
  float f = 0.555f;
  printf("%f", sqrt(f*4));
  return 0;
}
```